

Zusammenfassung DL4G

Deep Learning for Games

Maurin D. Thalmann

21. Januar 2020

Inhaltsverzeichnis

1 Sequential Games with perfect information	3
1.1 Finite Sequential Games	3
1.2 Complexity Factors in Game Analysis	3
1.3 Illustration of State Space Complexity	3
1.4 Extensive Form Representation	3
1.5 Game Tree Analysis - Backward Induction	4
1.6 Reasoning about Finite Sequential Games	4
1.7 Zero-Sum Games	5
1.8 Minimax Algorithm	5
1.9 Search Tree Pruning	6
1.10 Illustrations for Alpha-Beta Pruning	6
2 Monte Carlo Tree Search	7
2.1 Random Walks	7
2.2 The 4 Phases in Monte Carlo Tree Search	8
2.3 MCTS for Tic-Tac-Toe	8
2.4 Selection Policy: Which Node to Choose	9
2.5 Example of Multi-Armed Bandits	9
2.6 UCB1: Upper Confidence Bound	9
2.7 Minimax vs. Monte Carlo Tree Search	10
2.8 MCTS for Zero-Sum Games	10
3 Information Sets	11
3.1 Formal Definition of Information Sets	11
3.2 Perfect vs. Imperfect Information	11
3.3 Determinization	11
3.4 Strategy Fusion	12
3.4.1 Strategy Fusion Example	12
3.5 Information Set Search Trees	12
3.6 Information Set Monte Carlo Tree Search	13
3.7 Information Set vs. Perfect Information MCTS	14
4 Supervised Machine Learning	14
4.1 Disciplines in Machine Learning	14
4.2 Example Binary Classification Problem	14
4.3 Hyperparameters	16
4.4 Simplest Machine Learning Workflow	16
4.5 Train / Test Splits	16
4.6 How to get Fired as a Data Scientist	16
4.7 More Complex Evaluation Workflows	17
4.8 Confusion Matrix for Binary Classifiers	17
4.9 Accuracy and Error Rate	17
4.10 Multiclass Classification	18
5 Neuronal Networks	18
5.1 AI vs. Machine Learning vs. Deep Learning	18
5.2 Task Types	19
5.3 Feed forward Network	19
5.4 Linear Model	20
5.5 Neural network basics in Keras	20
5.6 Activation Function	20
5.7 Loss Function	21
5.8 Likelihood	21
5.9 Optimal loss function	22
5.10 Multi-class problem	22
5.11 Gradient descend	22

5.12 Back propagation	23
6 Deep Neuronal Networks	23
7 Convolutional Neuronal Networks	23
8 Reinforcement Learning	23

1 Sequential Games with perfect information

1.1 Finite Sequential Games

- Eine endliches Set an **Spielern**, jeder mit einem endlichen Set an möglichen **Aktionen**
- Spieler wählen ihre Aktionen **sequenziell** (einer nach dem anderen, in Zügen)
- Eine endliche Anzahl an **Zügen** wird gespielt
- Spätere Spieler **beobachten** die Züge der früheren Spieler (Perfect Recall)
- Eine **Strategie** sagt dem Spieler, welche Aktion er in seinem Zug spielen soll
- Ein **Strategieprofil** ist eine gewählte Strategie eines jeden Spielers
- Ein **Utility** oder **Payoff Function** bestimmt den Ausgang jedes Aktionprofils

1.2 Complexity Factors in Game Analysis

1. Anzahl Spieler
 - Spiele mit 4 Spielern sind schwieriger zu analysieren als solche mit 2 Spielern
2. Grösse des Suchraums
 - Bestimmt durch Anzahl gespielte Züge und Anzahl Aktionen für jeden Spieler
3. Kompetitive Spiele vs. Kooperative Spiele
 - Kompetitive Spiele involvieren Spieler mit komplett gegensätzlichen Interessen
4. Stochastische Spiele vs. Deterministische Spiele
 - Stochastische Spiele beinhalten Zufälle, bspw. Verteilung der Karten, Würfel rollen
5. Perfekte vs. imperfekte Informationsspiele
 - Imperfekte Information heisst das Spiel ist nur teilweise überwachbar, bspw. kennen wir nicht die Karten eines gegnerischen Spielers beim Poker oder Jass

1.3 Illustration of State Space Complexity

Game	State Space (as log to base 10; 10^x)
Tic-Tac-Toe	3
Connect-4	13
Backgammon	20
Chess	47
Go 19x19	170

- State Space beschreibt die Anzahl erlaubter Boardpositionen
- Schach / Chess hat 10^{47} verschiedene Boards, Go hat 10^{170} verschiedene Boards
- Zum Vergleich: Geschätzt sind im Universum 10^{80} Atome

1.4 Extensive Form Representation

- Sequenzielle Spiele können (im Prinzip) als Spielbäume repräsentiert werden
- Knoten sind Spielzustände/Positionen und Kanten sind Aktionen/Bewegungen
- Blätter (Leaves) bestimmen den Payoff

1.5 Game Tree Analysis - Backward Induction

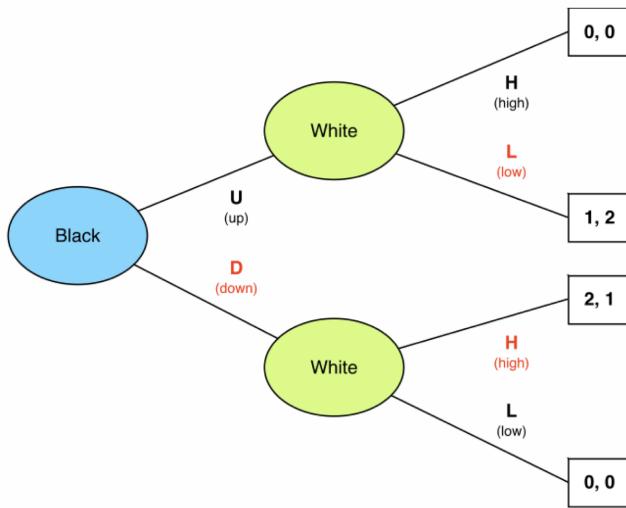


Abbildung 1: Backward Induction am Beispiel eines simplen Spielbaums

- Backward Induction ist der Lösungsalgorithmus für endliche, sequenzielle Spiele
- Im Beispiel oben hat Black den First-Mover Vorteil.
- Wenn beide Spieler perfekt spielen, endet das Spiel mit den Payoffs (2,1) (2 für Black, 1 für White)
- Solche Spiele immer rückwärts analysieren!

1.6 Reasoning about Finite Sequential Games

- Eine **ultra-schwache Lösung** beweist ob der erste Spieler aus der Initialposition gewinnen, verlieren oder unentschieden machen wird, in Annahme eines perfekten Spiels des Gegners
Im Beispiel: Schwarz kann einen Gewinn forcieren und hat demnach den First-Mover Vorteil
- Eine **schwache Lösung** bietet einen Algorithmus welcher ein komplettes Spiel an perfekten Zügen aus der Initialposition offenbart, in Annahme eines perfekten Spiels des Gegners
Schwache Lösung für das Spiel: Black spielt U, White spielt L, Black spielt D
- Eine **starke Lösung** bietet einen Algorithmus, welcher perfekte Züge aus jeder Position produzieren kann, auch wenn vorher von irgendeinem Spieler Fehler gemacht wurden.
Starke Lösung für dieses Spiel:
 - Algorithmus für White: wenn Black U spielt → spiel L; wenn Black D spielt → spiel H
 - Algorithmus für Black: Spiel D im ersten Zug; wenn White L im oberen Knoten spielt, spiel D

A Strong Solution to Nim Algorithmus für einen perfekten Nim Bot:

1. Wenn nur ein Haufen übrig bleibt
→ Nimm alle Objekte des Haufens und hol den Preis
2. Wenn zwei Haufen mit unterschiedlicher Anzahl Objekte übrig bleiben
→ Nimm Objekte vom grösseren Haufen und mache beide gleich gross
3. Wenn zwei Haufen diesselbe Anzahl Objekte haben
→ Egal was du tust, du hast verloren (in Annahme eines perfekten Spiels des Gegners)

1.7 Zero-Sum Games

- Ein Spiel ist Zero-Sum, wenn der totale Gewinn des Siegers gleich dem totalen Verlust des Verlierers ist
 - Einen Kuchen zu schneiden ist zero-sum bspw. wenn ich ein Stück esse, ist es für dich verloren
 - Brettspiele sind zero-sum bspw. wenn der Sieger +1 erhält und der Verlierer -1
- u_1 und u_2 umschreiben die Utility-Funktion von Spieler 1 und 2, somit ist für jedes Strategiepaar s_1 und s_2 von Spieler 1 und 2 und es gilt $u_1(s_1, s_2) + u_2(s_1, s_2) = 0$, dann lässt sich sagen:

$$u_1(s_1, s_2) = -u_2(s_1, s_2)$$

Backward Induction & Minimax

- Backward Induction ist die Lösungsstrategie für endliche Spiele mit perfekter Information
- Eine einzelne Durchführung von Backward Induction aus einem Startzustand offenbart eine schwache Lösung. Wenn Backward Induction dynamisch (während des Spiels) aus jedem Zustand ausgeführt werden kann, erhalten wir eine starke Lösung.
- Wenn das Spiel zusätzlich zero-sum ist, kann Backward Induction mit dem Minimax Algorithmus implementiert werden. Minimax wird oft für Zwei-Spieler-Spiele definiert, ist aber auch für mehr Spieler erweiterbar.
- Minimax erlaubt effizientes Pruning („Ausästen“) und nahtlose Integration von Heuristiken
- Hinweis: Backward Induction kann auch für Spiele genutzt werden, die nicht zero-sum sind und kompliziertere Payoffs enthalten als eine einzelne Zahl

1.8 Minimax Algorithm

- 1928 von John von Neumann erfunden
- Beide Spieler wollen ihre respektiven Payoffs maximieren
- Weil das Spiel zero-sum ist, ist mein Gewinn = Verlust des Gegners
- Anstatt nach eigenem Gewinn zu maximieren, kann der Gewinn des Gegeners minimiert werden
- Im Spielbaum kann folgendermassen vorgegangen werden:
 - Gehört der Knoten mir, wähle die Aktion welche den Payoff maximiert
 - Gehört der Knoten dem Gegner, wähle die Aktion welche den Payoff minimiert

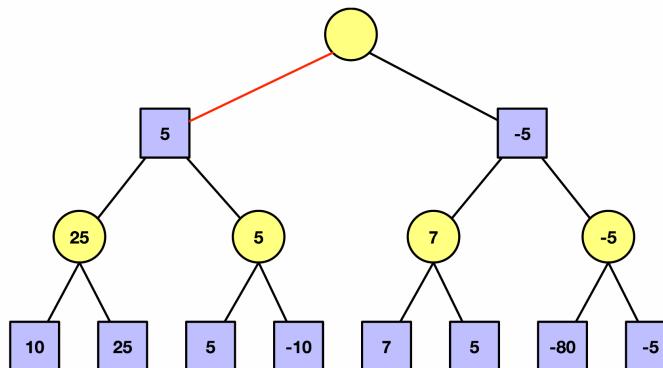


Abbildung 2: Lösungsweg eines Spielbaums mithilfe des Minimax-Algorithmus

Programming a Minimax Bot

- Minimax implementiert Backward Induction für zero-sum Spiele. Dank der vereinfachenden zero-sum Eigenschaft können einige Tricks angewandt werden, um einen heresischen Algorithmus zu erhalten:
 - Minimax nur bis zu einer limitierten Tiefe, bspw. 5 Runden vorausschauen und dann stoppen. Wie tief man gehen kann ist abhängig von den Spielregeln (Anzahl mögliche Züge), Effizienz der Implementation und Rechenleistung.

- Da die Tiefe limitiert ist, werden die Blattknoten nicht erreicht und wir kennen die echten Payoffs nicht. Wir müssen eine Heuristik erfinden, um die Situation abzuschätzen an welcher wir stoppen. Je besser die Heuristik, desto besser der Bot.

```

function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        bestValue := - $\infty$ 
        for each child of node
            val := minimax(child, depth - 1, FALSE)
            bestValue := max(bestValue, val)
        return bestValue
    else
        bestValue := + $\infty$ 
        for each child of node
            val := minimax(child, depth - 1, TRUE)
            bestValue := min(bestValue, val)
        return bestValue

```

Abbildung 3: Pseudocode eines Minimax-Algorithmus mit limitierter Tiefe und Heuristik

1.9 Search Tree Pruning

- Es müssen nicht alle Knoten abgelaufen werden, um die optimale Strategie zu finden.
- Wir „stutzen“ (prunen) Sub-Bäume, welche keine bessere Lösung beinhalten können und demnach nicht besucht werden müssen.
- Dazu enthält der Algorithmus zwei Parameter:
(Vorfahren sind alle Knoten auf dem Weg zwischen dem aktuellen und dem Wurzelknoten)
 - α ist der höchste Wert aller MAX-Vorfahren eines MIN Knoten
 - β ist der tiefste Wert aller MIN-Vorfahren eines MAX Knoten
- Der Algorithmus Alpha-Beta Pruning aktualisiert diese beiden Parameter im Minimax-Prozess und schneidet nicht besuchte Sub-Bäume ab, sobald er weiß dass die Werte aus diesem Sub-Baum den Wert α nicht überbieten oder β nicht unterbieten können.

Alpha-Beta Pruning Rules

- Alpha (α) ist der minimale Score, welcher dem maximierenden Spieler versichert werden kann
- Beta (β) ist der maximale Score, welcher dem minimierenden Spieler versichert werden kann
- Daraus lassen sich folgende beiden Regeln schliessen:
 - **Regel 1:** Schneide ab, sobald der aktuelle Wert eines MIN Knoten kleiner ist als α
 - **Regel 2:** Schneide ab, sobald der aktuelle Wert eines MAX Knoten grösser ist als β

1.10 Illustrations for Alpha-Beta Pruning

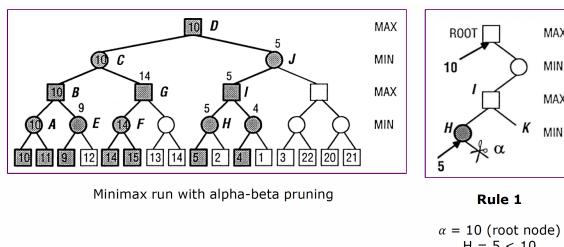


Abbildung 4: Illustration der Durchführung der 1. Regel des Alpha-Beta Pruning

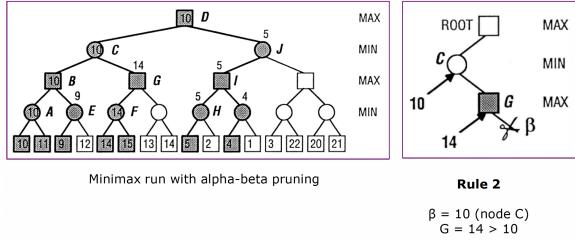


Abbildung 5: Illustration der Durchführung der 2. Regel des Alpha-Beta Pruning

```

function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        for each child of node
            α := max(α, alphabeta(child, depth - 1, α, β, FALSE))
            if β ≤ α
                break (* β cut-off *)
        return α
    else
        for each child of node
            β := min(β, alphabeta(child, depth - 1, α, β, TRUE))
            if β ≤ α
                break (* α cut-off *)
        return β
    
```

Abbildung 6: Pseudocode eines Minimax mit limitierter Tiefe mithilfe von Alpha-Beta Pruning

Speed-Up of Alpha-Beta Pruning

- In einem Spielbaum mit Tiefe m mit b möglichen Aktionen bei jedem Knoten ist die Zeitkomplexität des Minimax $O(b^m)$ bzw.. es gibt b^m Blattknoten
- Im Idealfall benötigt Alphe-Beta Pruning nur $O(b^{m/2}) = O((\sqrt{b})^m)$. Dies korrespondiert zu einer Reduzierung des Branching-Faktors von b zu \sqrt{b} , bspw. bei Schach bedeutet dies 6 mögliche Aktionen bei jedem Knoten (anstelle von 35)
- Um diesen maximalen Speed-Up zu erreichen, müssen die verschiedenen States in gescheiter Anordnung erforscht werden, was jedoch problemspezifisch ist.

2 Monte Carlo Tree Search

2.1 Random Walks

- Suchräume sind meist zu gross für eine vollständige Suche
- Minimax soll bei einer bestimmten Baum-Tiefe stoppen und raten (mit Heuristik)
- Monte Carlo Tree Search ist ein anderes Vorgehen:
Monte Carlo Tree Search führt Random Walks durch, um möglichst viel des Suchbaums in einem vorbestimmten Zeitraum abzusuchen. Danach wird der vielversprechendste Zug gespielt.

2.2 The 4 Phases in Monte Carlo Tree Search

1. **Selection**
 - Starte beim Wurzelknoten R und wähle fortlaufend Kinderknoten
 - Stoppe, wenn du einen Knoten erreichst, der noch nicht komplett erweitert/erforscht wurde
 - Benötigt ein Kriterium für die Auswahl der Kinderknoten, sogennante *tree policy*
2. **Expansion**
 - Wenn das Zeitlimit L das Spiel beendet, gib die Payoffs zurück
 - Sonst, wähle eine unerforschte Aktion und kreiere einen Knoten C für diese
3. **Simulation**
 - Simuliere ein Weiterspielen von Knoten C aus, mithilfe einer *default policy*
 - Im simpelsten Fall, spiele einfach bis zu irgendeinem Ende mit zufälligen Zügen
4. **Backpropagation**
 - Aktualisiere die gespeicherten Informationen in jedem Knoten von C zurück bis zu R
 - MCTS erwartet einen Payoff in $[0,1]$

2.3 MCTS for Tic-Tac-Toe

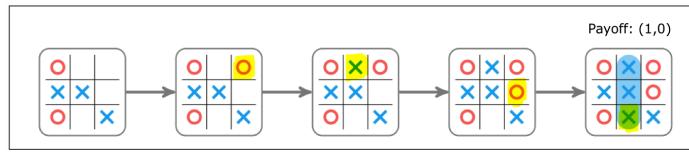


Abbildung 7: Zufälliger Durchlauf eines Runden Tic-Tac-Toe

- Es gibt Punkte für einen Gewinn (1), unentschieden (0.5) und Verlust (0)
- MCTS ist nicht limitiert auf zero-sum Spiele
- Payoffs werden durch Vektoren repräsentiert
- Die Simulation endet mit einem Gewinn für Spieler X mit Payoff +1 → Payoff-Vektor (1,0) (Spieler Y hat in dieser Simulation also einen Payoff von 0)

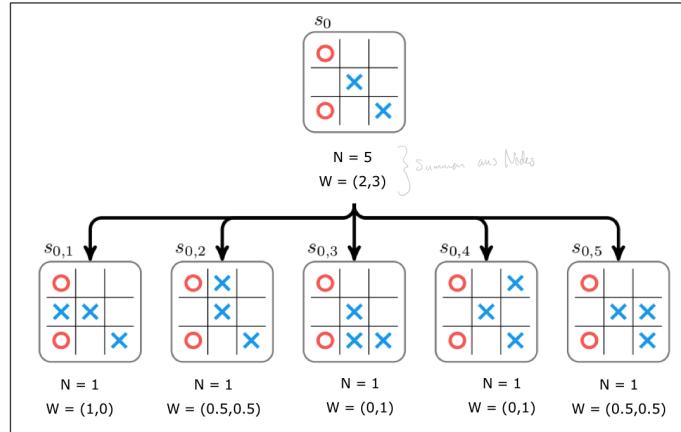


Abbildung 8: Übersicht der Simulationen eines Tic-Tac-Toe-Spiels

- N speichert die Anzahl Simulationen, die vom Knoten s_0 aus gestartet wurden
- W sind die akkumulierten Payoff-Vektoren (eine Komponente für jeden Spieler)
- Backpropagation kalkuliert lediglich die Summe von N^* und W
- **Wichtig:** Ist der Wurzelknoten ebenfalls ein Kindesknoten, muss sein Payoff nach oben ebenfalls hinzugerechnet werden

2.4 Selection Policy: Which Node to Choose

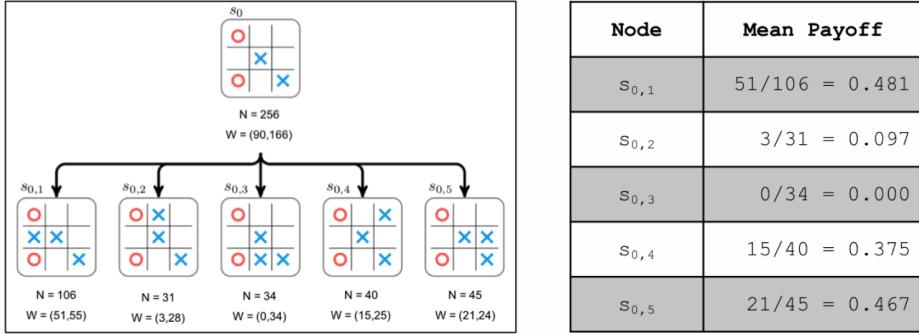


Abbildung 9: Durchschnittliche Payoffs der durchlaufenen Simulationen mit MCTS

- **Exploitation**

Knoten $s_{0,1}$ hat den höchsten durchschnittlichen Payoff bzw. basierend auf aktuell verfügbaren Informationen, maximiert dieser Knoten meinen erwarteten Gewinn

- **Exploration**

Knoten $s_{0,4}$ wurde nur 40 mal probiert, wie kann ich sicher sein dass der Score tiefer bleibt als bei $s_{0,1}$, auch wenn ich jetzt noch 66 mal spielen würde?

2.5 Example of Multi-Armed Bandits

- Es gibt eine Reihe an Slotmaschinen mit verschiedenen (unbekannten) Auszahlungswahrscheinlichkeiten und -mengen
- Man hat 1000 Münzen und möchte den erwarteten Gewinn erhöhen
- Idealerweise würde man immer an der Maschine mit dem grössten erwarteten Gewinn spielen
- Unglücklicherweise weiss man jedoch nicht, welche Maschine dafür am besten ist
- Es ist keine Person da, welcher man zuschauen und mehr über die Maschinen erfahren kann

Wie wählt man nun die beste Strategie in dieser Situation?

2.6 UCB1: Upper Confidence Bound

- Am besten Exploration und Exploitation ausbalancieren
 - **Exploration:** Spiele alle Maschinen um möglichst viele Informationen zu sammeln
 - **Exploitation:** Spiele die beobachtet beste Maschine um den erwarteten Gewinn zu maximieren
- UCB1 bietet die beste Balance zwischen Exploration und Exploitation, es gibt quasi ein statistisches Konfidenzintervall für jede Maschine aus
 - Parameter $c \geq 0$ kontrolliert den Trade-Off zwischen Exploitation (tiefes c) und Exploration (hohes c)

$$U_i = \frac{W_i}{N_i} + c \sqrt{\frac{\ln N_p}{N_i}}$$

$\frac{W_i}{N_i}$ **Exploitation**, der durchsch. Gewinn für Maschine i ($\frac{W(\text{Gewinne})}{N(\text{Versuche})}$)

$\sqrt{\frac{\ln N_p}{N_i}}$ **Exploration**

N_p Wie oft haben wir insgesamt gespielt (wie viele Münzen wurden verbraucht)

N_i Wie oft wurde Maschine i ausprobiert

How to play with UCB1 Für jede der 1000 Münzen...

- Kalkuliere das UCB1 (U_i) von jeder Maschine i
- Spiele an der Maschine mit dem höchsten Upper Bound (U_i)
- Wähle entweder zufällig oder in numerischer Ordnung
- Solange man dies tut, wird der beobachtete Mittelwert für die Maschine sich verschieben und das Konfidenzintervall wird schmäler, aber die Intervalle aller anderen Maschinen werden breiter. Der Upper Bound einer anderen Maschine wird womöglich grösser als jener der aktuellen Maschine und man wird zu dieser Maschine wechseln.
- Bei Berechnung des UCB1 muss die Vektorkomponente für den aktuellen Spieler gewählt werden!
Bsp. $UCB1(s_{0,1}) = 0.71$, $UCB1(s_{0,2}) = \dots$

The final Move

- Wenn die Zeit vorüber ist, spiele die erste Aktion mit der **höchsten Anzahl an Besuchen N !**
 - Es benötigt keine Exploration wenn die finale Aktion ausgewählt werden soll
 - Deshalb ist die Aktion mit dem höchsten UCB1 Score in der letzten Runde nicht zwingend die beste Wahl
 - Die Aktion, welche am meisten gewählt wurde, gab meistens auch den höchsten UCB1 Score

2.7 Minimax vs. Monte Carlo Tree Search

- Beide Algorithmen setzen perfekte Information voraus
- Minimax ist nur für 2-Spieler zero-sum Spiele anwendbar
- MCTS funktioniert für jedes perfekte-Informationsspiel
- Minimax optimiert Payoffs; MCTS optimiert einen Exploitation-Exploration Trade-Off
- In Kontrast zu Minimax, ist MCTS ein „Anytime“ Algorithmus
- Monte Carlo Bäume sind asymmetrisch, Minimax Bäume sind symmetrisch

2.8 MCTS for Zero-Sum Games

- Im Fall von 2 Spielern, Payoff Vektoren sind von der Form $(W, N - W)$
- Spieler 1 maximiert W , Spieler 2 maximiert $N - W$ (ignoriert den Exploitation-Teil in UCB1)
- Anders gesagt, könnte Spieler 2 $-W$ minimieren
- Wir können nur die Rewards W für Spieler 1 speichern und die Zeichen für Spieler 2 „flippen“
- Daraus entsteht eine Minimax-ähnliche Variante von MCTS

Convergence to Optimal Play

- Mit genügend Ressourcen konvergiert MCTS zu Minimax
- UCB1 hat logarithmisches Bereuen (*logarithmic regret*), weil nur bestimmte Zeit zur Verfügung steht
- Die Fehlerwahrscheinlichkeit an der Wurzel des Baums konvergiert zu null in polynomialer Rate wenn die Anzahl simulierter Spiele in Richtung unendlich ansteigt. Dieser Beweis bedeutet, unter Annahme von genug Zeit, dass UCB1 der MCTS erlaubt zum Minimax-Baum zu konvergieren und ist demnach optimal.

3 Information Sets

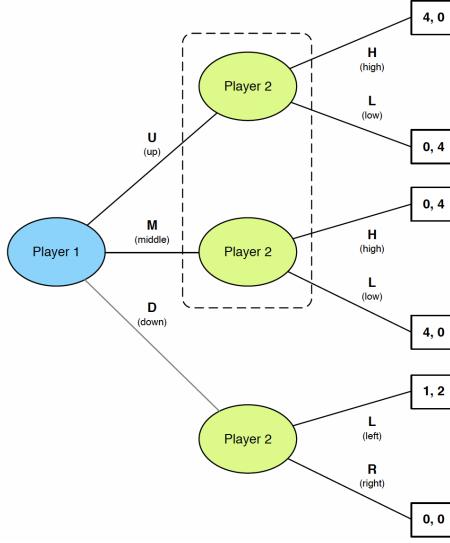


Abbildung 10: Spielbeispiel mit imperfekter Information für Spieler 2

- In diesem Beispiel kann Spieler 2 nicht zwischen den oberen beiden Zuständen unterscheiden (gestrichelte Box), also ob Spieler 1 im ersten Zug U oder M gewählt hat.
- Er weiss also auch nicht, ob Spieler 1 D gewählt hat oder nicht
- Spieler 1 kann also zufällig zwischen U und M wählen

3.1 Formal Definition of Information Sets

- Ein Information Set von Spieler i ist eine Sammlung von Knoten, welche Spieler i gehören, zwischen welchen ich nicht unterscheiden kann.
- Es müssen einige Regeln erhoben werden, wie ein Information Set aufgebaut sein sollen. Ansonsten könnte Spieler 2 betrügen (bspw. mithilfe von Side-Channel-Informationen). Wir gehen jedoch von Perfect Recall aus, dass sich jeder Spieler seine bisherigen Züge merken kann.
 1. Alle Knoten in einem Information Set müssen die gleichen Aktionen / Kinder haben.
 2. Aufgrund meiner Historie (bzw. vorgängigen Aktionen) darf ich keinen Hinweis darauf erhalten, dass ich mich in einem Information Set befinde.

3.2 Perfect vs. Imperfect Information

- In einem perfekten Informaionsspiel besteht jedes Information Set aus genau einem Knoten. Ansonsten ist es ein Spiel mit imperfekter Information.
- Eine (pure) Strategie von Spieler i ist ein kompletter Plan von Aktionen. Sie spezifiziert, was Spieler i tun wird, für jedes seiner Information Sets.
- Backward Induction kann nicht für Spiele mit imperfekter Information angewandt werden
- Stattdessen müssen sequenzielle Spiele mit imperfekter Information in simultane Spiele umgewandelt und mit der Best Response Funktion ein Gleichgewicht gefunden werden.

3.3 Determinization

- Determinization erlaubt, alle Möglichkeiten für Spiele mit perfekter Information für Spiele mit imperfekter Information nutzen zu können.
- Eine Determinization ist eine Probe aus dem Information Set des aktuellen Spielstands, bspw. für Kartenspiele, ziehe zufällig eine Handvoll Karten an Spieler, welche konsistent ist mit den Beobachtungen des aktuellen Spielers
- Zum Beispiel:

- Wende MCTS auf jede Determinization an
- Wähle einen Zug, für welchen die summierte Anzahl Besuche über alle Bäume hinweg maximal ist

3.4 Strategy Fusion

Strategy Fusion referenziert auf den Effekt, wenn ein deterministischer „Solver“ unterschiedliche Entscheidungen innerhalb desselben Information Set macht

3.4.1 Strategy Fusion Example

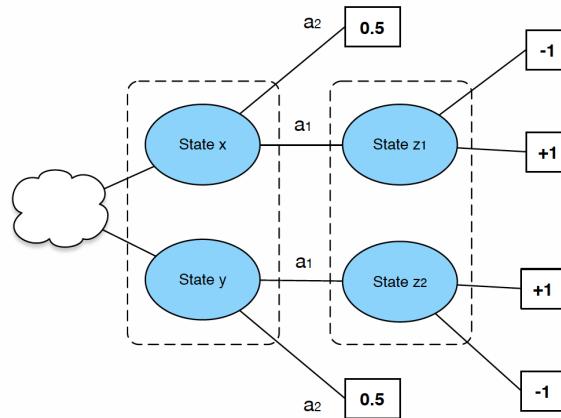


Abbildung 11: Strategy Fusion an einem Information Set Beispiel

- 1-Spieler-Spiel mit zwei Information Sets
- Spieler kann a_2 wählen für einen sofortigen Payoff von 0.5
- Wählt er a_1 , ist der erwartete Payoff 0
- Die beste Strategie ist demnach a_2 zu spielen
- Determinization würde von x,y sampeln und einen perfekten-Informations-Gewinn spielen
- Aber Sampling vom ersten Information Set würde das zweite Information Set auflösen (unter Annahme von Perfect Recall)
- In einem perfekten Informationsspiel würde der Spieler bei der Wahl von a_1 immer einen Payoff von +1 erhalten
- Determinization würde hier fälschlicherweise empfehlen, a_1 zu spielen
- Dies röhrt daher, dass wir mit perfekter Information den besten Spielzug von z_1 und z_2 bestimmen könnten

3.5 Information Set Search Trees

Strategy Fusion bewältigen, indem ein Baum von Information Sets durchsucht wird, anstelle von Zuständen. (Sampling von x,y enthüllt nicht direkt z_1 oder z_2)

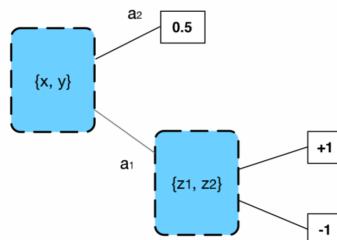


Abbildung 12: Baum von Information Sets zur Durchsuchung

Applied to Jass

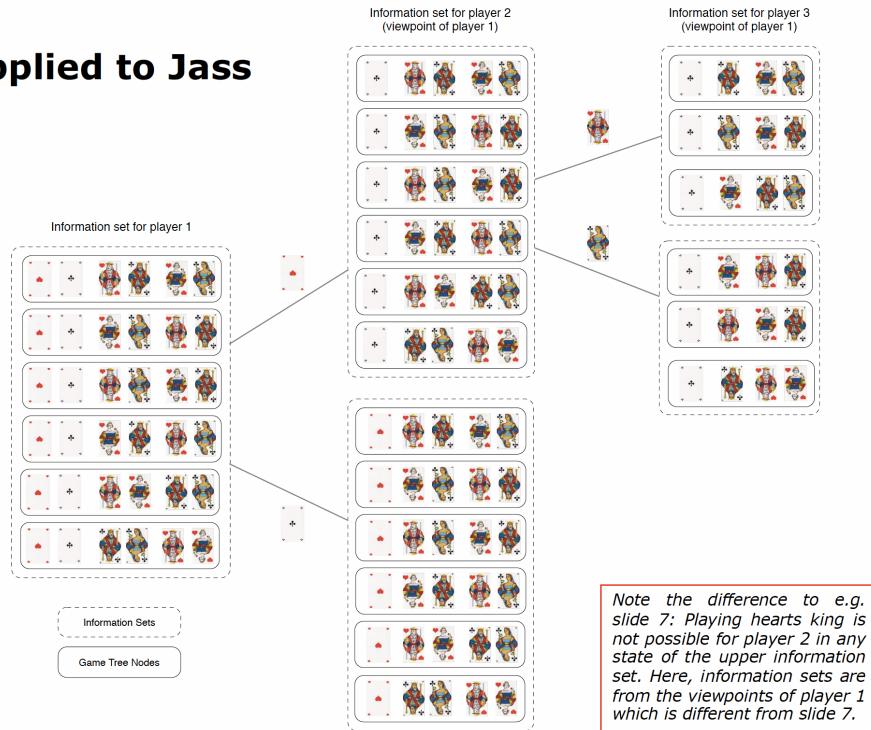


Abbildung 13: Information Set Search Trees am Beispiel von Jass

3.6 Information Set Monte Carlo Tree Search

- Knoten korrespondieren zu Information Sets aus der Sicht des Root-Spielers
- Kanten korrespondieren zu Aktionen (welche in mindestens einem Zustand verfügbar sind)
- Der Algorithmus beginnt damit, vom Root Information Set zu samplen und blendet alle inkompatiblen Knoten mit der gesampleten Determinization aus
- Dann wird MCTS auf dem perfekten Informationsspiel ausgeführt

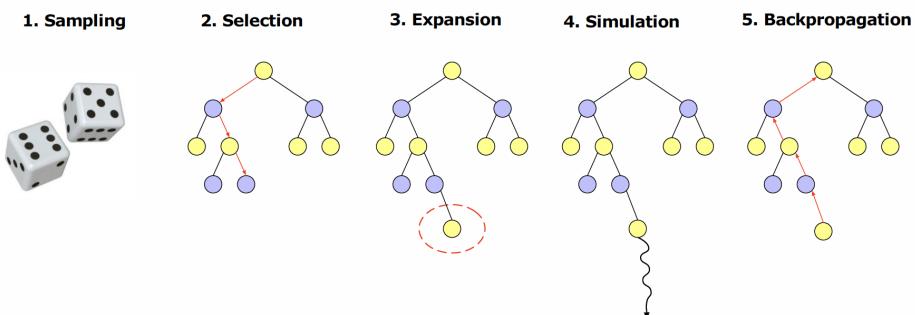


Abbildung 14: Phasen der Information Set Monte Carlo Tree Search

Necessary Change in Selection Phase

- Für die Selection wird UCB1 verwendet:

$$U_i = \frac{W_i}{N_i} + c \sqrt{\frac{\ln N_p}{N_i}}$$
- Bei ISMCTS kann sich die Anzahl möglicher Züge an einem gegnerischen Knoten zwischen den Besuchen an diesem Knoten unterscheiden
- Dies korrespondiert zum Subset-Armed-Bandit-Problem:
Ersetze N_p mit der Anzahl Besuchen des Vater-Knotens und wie oft Knoten i verfügbar war für die Selection
- Ohne diese Modifizierung ↑ werden seltene Züge „übererforscht“

3.7 Information Set vs. Perfect Information MCTS

- ISMCTS hat den Vorteil, dass ganze „rechnerische Budget“ auf einen einzelnen Baum konzentriert wird. Die Determinization-Technik hingegen teilt dies auf mehrere Bäume auf. Dies erlaubt üblicherweise eine tiefere Suche.
- Jedoch kann für einige Spiele, der Branching Faktor drastisch steigen aufgrund einer hohen Anzahl möglicher Züge bei jedem Information Set. Dadurch sinkt die Performance von ISMCTS in den meisten Fällen im Vergleich zur Determinization.

4 Supervised Machine Learning

4.1 Disciplines in Machine Learning

1. Supervised Learning
 - Dem Algorithmus werden gelabelte Trainingsdaten übergeben
 - Er lernt, Labels von bisher unbekannten Beispielen voraussagen zu können
2. Unsupervised Learning
 - Dem Algorithmus werden ungelabelte Daten übergeben
 - Er erkennt und exploitiert die Struktur der Daten
3. Semi-Supervised Learning
 - Eine Mischung aus supervised und unsupervised ML-Techniken
 - Wird meist dann genutzt, wenn nur limitiert gelabelte Daten vorhanden sind
4. Reinforcement Learning
 - Keine Daten verfügbar, aber der Algorithmus wird von einer Reward Funktion geleitet
 - Es sucht das ideale Verhalten, welches den Reward des Agents maximiert

4.2 Example Binary Classification Problem

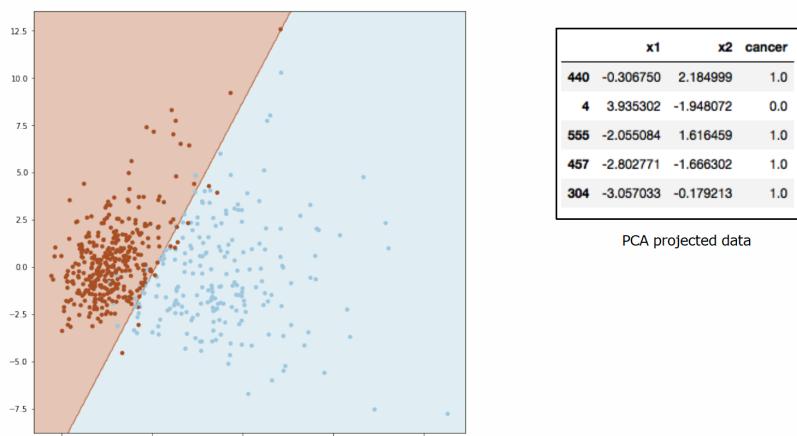


Abbildung 15: Decision Boundary am Beispiel von Brustkrebsanalysen

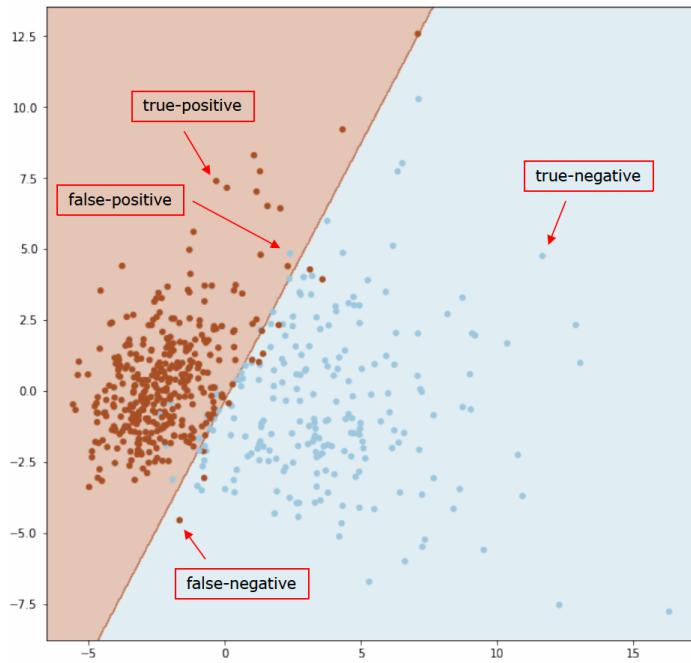


Abbildung 16: Beispiel der Confusion Matrix an Beispiel aus Abbildung 15

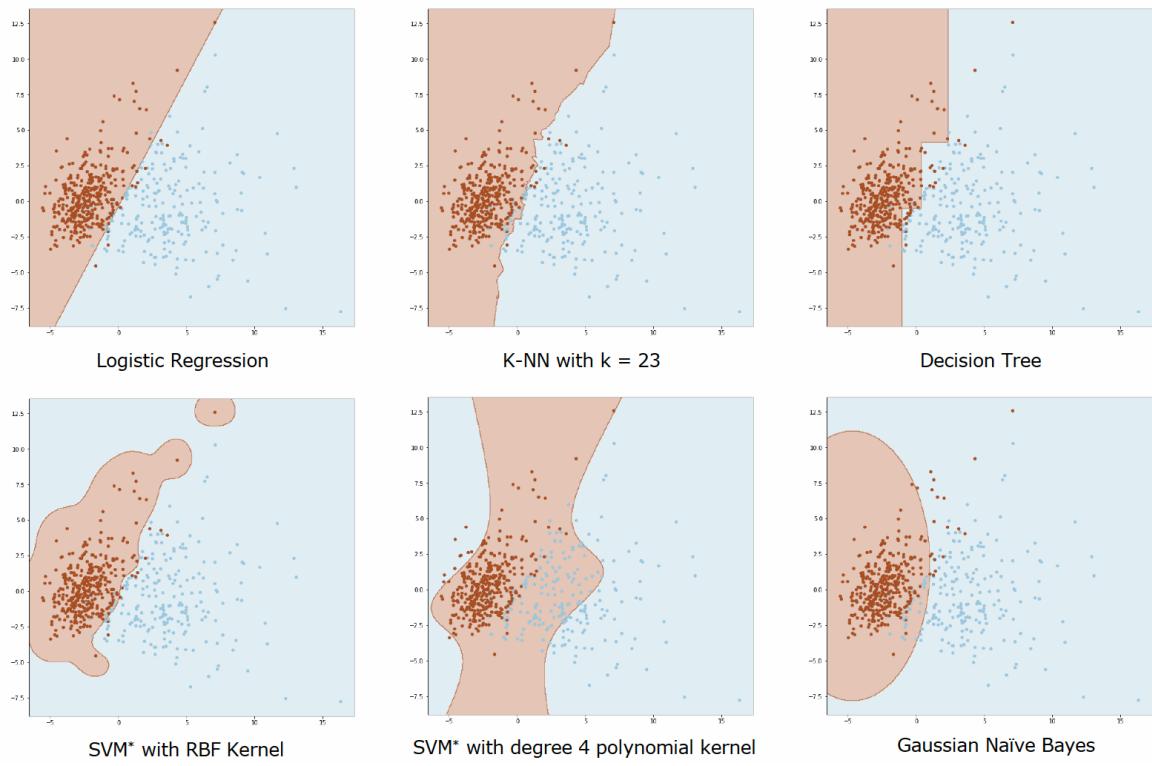


Abbildung 17: Unterschiede zwischen verschiedenen Classification Algorithmen

4.3 Hyperparameters

Hyperparameter stellen die manuelle Konfiguration von Machine Learning Modellen dar. Beispiele von solchen Hyperparametern:

- In k-NN ist die Anzahl Nachbarn k ein Hyperparameter
- Grad des Polynoms für Regressionsmodelle
- Regularisierungsparameter
- Kernel für Support-Vektor-Maschinen (SVM)
- Tiefe des Baumes und Variablen-Selektions-Policy in Decision Tree Modellen
- Anzahl Layers, Neuronen, Aktivierungsfunktion in Deep Learning Modellen
- Dropout, Batch Normalisierung, Optimizer etc. in Deep Learning Modellen

Wenn man als ML-Ingenieur die Hyperparameter-Konfiguration erreichen möchte, welche auf unbekannten Daten am besten performt, kann eine simplistische Train-Test Aufteilung nicht genutzt werden.

4.4 Simplest Machine Learning Workflow

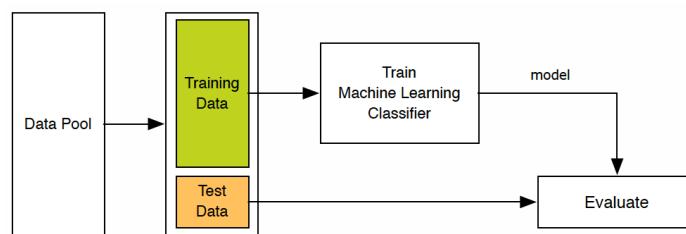


Abbildung 18: Einfachst möglicher ML Workflow

- Daten aufteilen in Trainings- und Test-Set
- Trainiere einen Klassifikator auf dem Trainings-Set
- Evaluation gibt Rückmeldung über Performance auf einem unbekannten Test-Set
- Dies funktioniert nur mit sehr viel Daten und wenn die Hyperparameter fixiert sind

4.5 Train / Test Splits

- Daten durchmischen (ausser man arbeitet mit Zeitreihen)
- Teile die Daten auf in Training- und Test-Set
- Empfohlene Aufteilung: 70% Training / 30% Test
- Training- und Test-Set müssen „disjoint“ sein (keine gemeinsamen Elemente haben)
- Baue dein Model auf Basis des Training-Set
- Evaluiere dein Model auf dem Test-Set
- Wenn das Test-Set nicht mehrmals genutzt wurde, gibt es eine Schätzung, wie gut das Modell auf unbekannten Daten performt

4.6 How to get Fired as a Data Scientist

- Anzahl Nachbarn k ist ein Hyperparameter
- Wir optimieren k indem wir testen: $k = 1, \dots, 10$
- Für jedes k wird...
 - das Modell auf dem Training-Set trainiert
 - die Performance auf dem Test-Set gemessen
- Hyperparameter $k = 6$ gewinnt, weshalb wir eine Accuracy von 0.99 auf unbekannten Daten erwarten → **Diese Schlussfolgerung ist gottverdammmt noch mal falsch!**
 - Der Hyperparameter ist gewählt basierend auf der Test-Set-Evaluation, demnach basiert dieser logischerweise auf dem Test-Set
 - Das Test-Set Resultat kann nicht mehr als unverfälschte Schätzung angesehen werden

4.7 More Complex Evaluation Workflows

- Empfohlene Aufteilung: 60% Training / 20% Validation / 20% Test
- Wie evaluiert man einen ML Klassifizierer korrekt:
 1. Loop über alle Hyperparameter-Kombinationen, die man ausprobieren möchte
 2. Trainiere das Modell mit ausgewählten Hyperparameter-Einstellungen auf dem Training-Set
 3. Evaluiere das Modell auf dem Validation-Set und messe die Performance
 4. Wähle das Modell mit der besten Performance als Kandidaten
 5. Evaluiere das Modell auf dem Test-Set für eine finale Performance-Schätzung
- Ein solcher Workflow benötigt eine grosse Menge an „guten“ Daten

4.8 Confusion Matrix for Binary Classifiers

Wir stellen uns einen Klassifizierer vor, welcher die Präsenz von Lungenkrebs voraussagen soll:

n=165	Predicted: NO	Predicted: YES
Actual: NO	50	10
Actual: YES	5	100

Das Test-Set besteht aus 165 Patientenaufzeichnungen ($n = 165$)

- 50x wurde vom Klassifizierer NO vorausgesagt, die wahre Aussage war tatsächlich NO
→ Der Klassifizierer hatte recht, dies ist ein **true-negative (TN)**
- 10x wurde vom Klassifizierer YES vorausgesagt, die wahre Aussage war aber NO
→ Der Klassifizierer lag falsch, dies ist ein **false-positive (FP)**
- 5x wurde vom Klassifizierer NO vorausgesagt, die wahre Aussage war aber YES
→ Der Klassifizierer lag falsch, dies ist ein **false-negative (FN)**
- 100x wurde vom Klassifizierer YES vorausgesagt, die wahre Aussage war tatsächlich YES
→ Der Klassifizierer hatte recht, dies ist ein **true-positive (TP)**

4.9 Accuracy and Error Rate

n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Accuracy misst, wie oft der Klassifizierer richtig lag.

(In diesem Fall: Accuracy = 91%, Error Rate = 9%)

- Accuracy = $\frac{TP+TN}{Total}$
- Error Rate = $\frac{FP+FN}{Total} = 1 - Accuracy$

Accuracy for Imbalanced Data

- Man stelle sich mal vor, man möchte eine seltene Krankheit voraussagen
- Ein Test-Set mit 5000 NO- und 20 YES-Aufzeichnungen ist nicht ungewöhnlich...
- Der Klassifizierer zeigt eine fantastische Accuracy von 99.6% - stimmt? Wohl kaum lol
- → Immer auf Klassen-Unausgeglichenheit überprüfen vor Berechnung der Accuracy!

4.10 Multiclass Classification

1. Einige Algorithmen sind inhärent Multiclass, wie Neurale Netzwerke oder Entscheidungsbäume
2. Für rein binäre Klassifizierer können wir one-vs-rest Techniken anwenden
 - Isoliere Diamanten gegen den Rest
 - Isoliere Herzen gegen den Rest
 - Zu guter Letzt, wähle die Klasse wessen Klassifizierer den höchsten Confidence-Score aus-gibt
3. Eine dritte Option wäre hierarchische Klassifizierung
 - (a) Der erste Klassifizierer entscheiden zwischen „Schieben“ oder „nicht Schieben“
 - (b) Der zweite Klassifizierer entscheiden zwischen Trumpf oder Obe-Abe, Une-Ufe
 - (c) etc.

		Predicted by Classifier						
		Diamonds	Hearts	Spades	Club	Obe-Abe	Une-Ufe	Push
Actual	Diamonds							
	Hearts							
	Spades							
	Club							
	Obe-Abe							
	Une-Ufe							
	Push							

Abbildung 19: Beispiel einer Multiclass Confusion Matrix am Beispiel vom Jassen

5 Neuronal Networks

5.1 AI vs. Machine Learning vs. Deep Learning

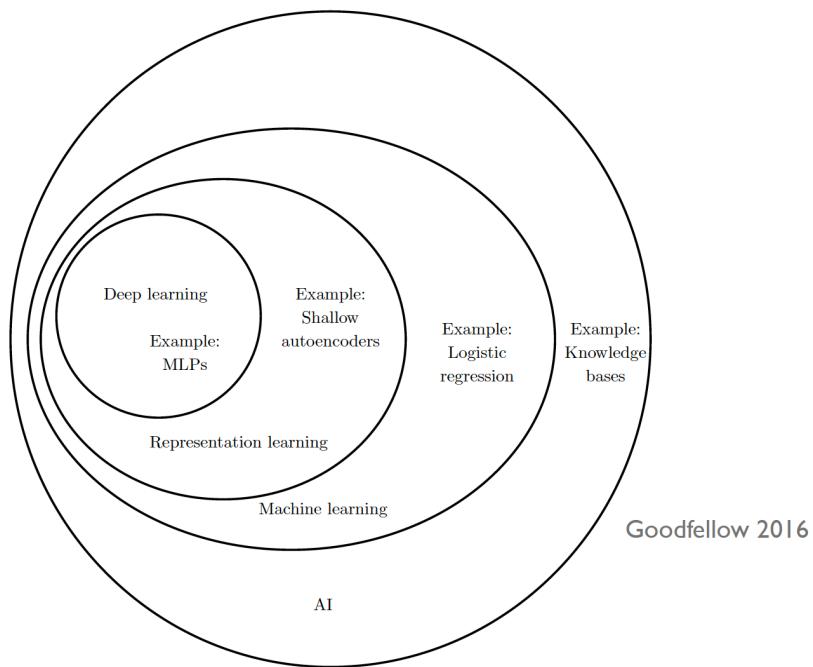


Abbildung 20: Darstellung der Unterschiede zwischen AI / ML / DL

Definition von Machine Learning Ein Computerprogramm **lernt** aus der Experience E mit Rücksicht auf eine Klasse von Tasks T und Performance Measures P , wenn es Tasks in T durchführt, gemessen an P , seine Experience E verbessert.

5.2 Task Types

- Regression: $f : \mathbb{R}^n \rightarrow \mathbb{R}$
- Classification: $f : \mathbb{R}^n \rightarrow 1, \dots, k$

5.3 Feed forward Network

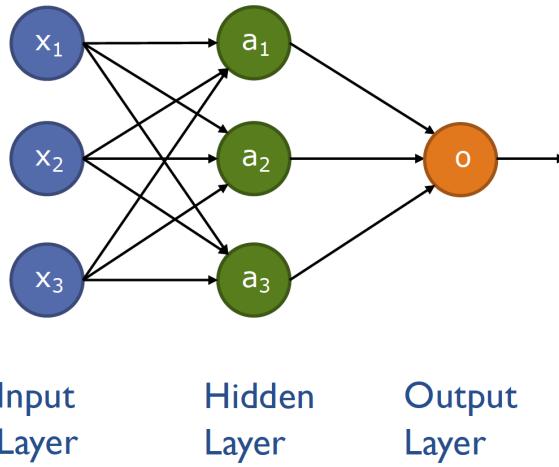


Abbildung 21: Feed Forward Network mit seinen verschiedenen Layers

- Jeder Knoten n_i im Netzwerk hat...
 - Inputs $\bar{x} = (x_1, \dots)$
 - Interne Parameter Θ
 - Eine Funktion $y = f_{n_i}(\bar{x})$, welche den Output berechnet
(Die Funktion wiederum ist abhängig von den Parametern Θ)

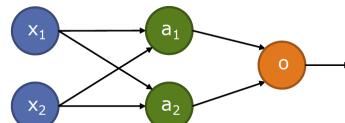
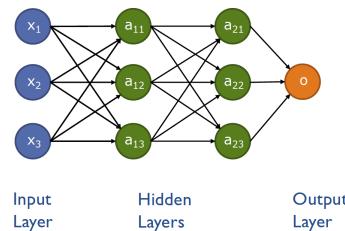


Abbildung 22: Feed Forward Network mit kleiner Anzahl Knoten

Recursive Calculation für ein Feed Forward Network:

$$y = f_0(a_1, a_2) = f_0(f_{a_1}(x_1, x_2), f_{a_2}(x_1, x_2))$$

Tiefere neurale Netzwerke haben lediglich mehr Hidden Layers (we need to go deeper $\rightarrow a_{n1}, a_{n2}, \dots$)



5.4 Linear Model

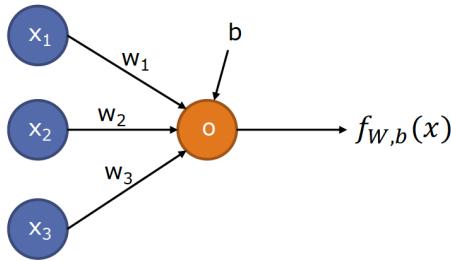


Abbildung 23: Linear Modell eines linearen Neuralen Netzwerks

$$f_{W,b}(x) = \text{act}(W^T x + b)$$

5.5 Neural network basics in Keras

Nein, hier sind keine Code-Beispiele aus dem Jupyter-Notebook zu finden. Installiert es selber und schaut euch das Beispiel aus dem Unterricht an.

- Lernen wird in *Batches* vollzogen
- Eine *Epoche* ist ein Training über das gesamte (Training-)Daten-Set, die Anzahl Epochen hängt vom Task, der Netzwerkkapazität und den Daten ab.
- Keras ist ein simples Interface für Beispiele und kleine Tasks, grössere Datensets benötigen meist detailliertere Kontrolle
- Keras ist das Standard High-Level Interface von Tensorflow

5.6 Activation Function

- Sigmoid ist eine gute Wahl für binäre Klassifizierung
- Dazwischenliegende Knoten können andere Aktivierungsfunktionen benutzen
- Aktivierungsfunktionen sollten non-linear sein
- Heutzutage (oder heute?) wird vor allem *relu* als Aktivierungsfunktion für Hidden Layers genutzt

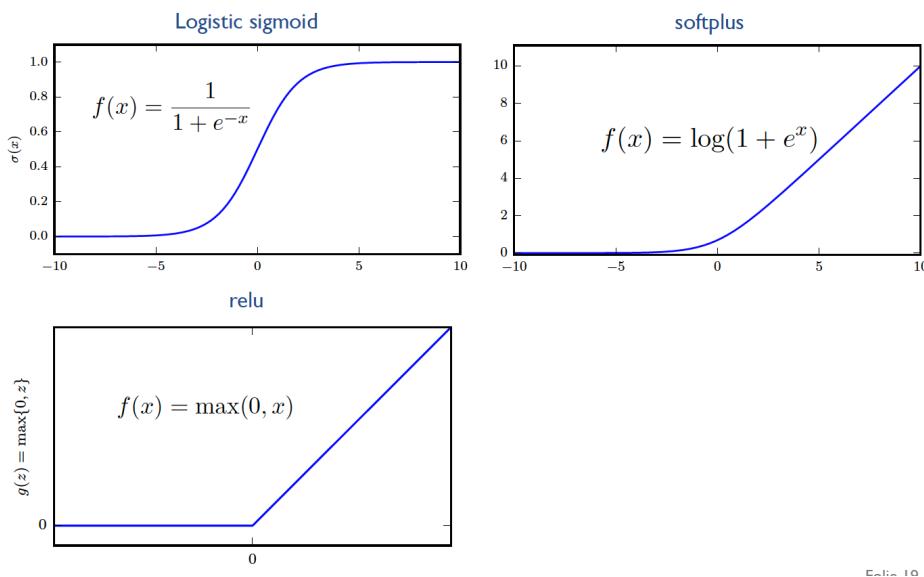
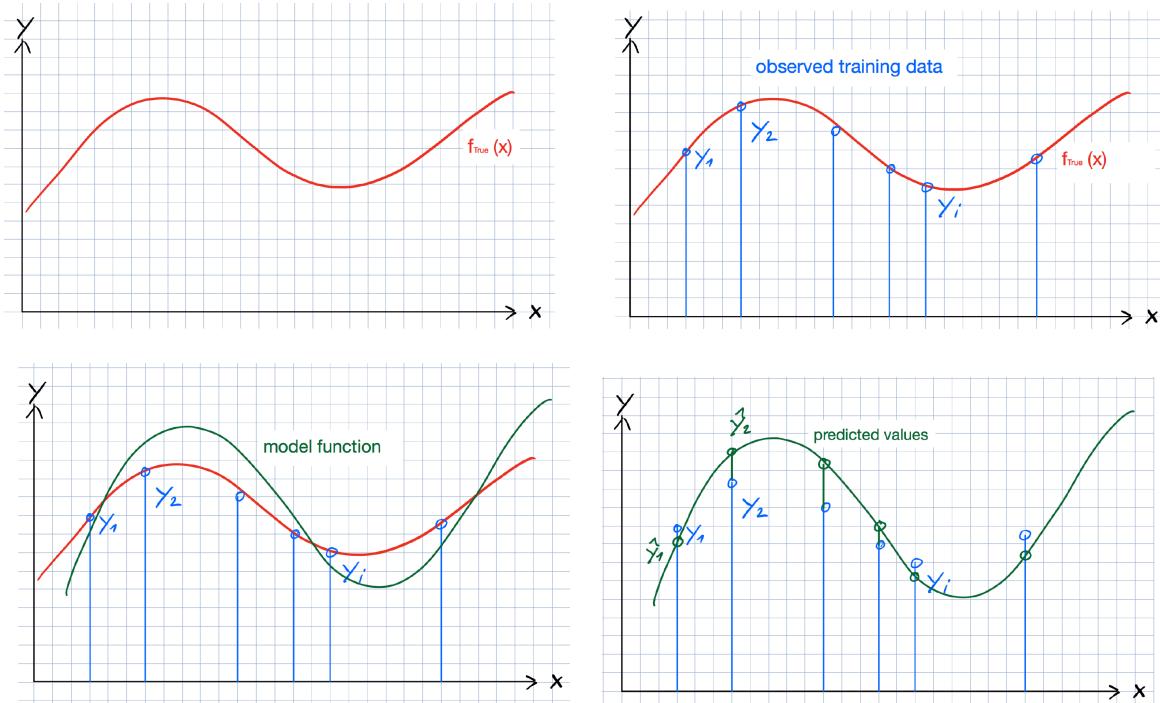


Abbildung 24: Verschiedene Aktivierungsfunktionen und deren Kurven

5.7 Loss Function

Loss (oder Kosten-)Funktion ist jene Funktion, welche von den Learning Tasks minimiert werden soll.

5.8 Likelihood



Was ist die Wahrscheinlichkeit (likelihood), dass die Daten (blaue Kurve) vom Modell (grüne Kurve) produziert wurden? Wähle ein Modell, welches diese Likelihood maximiert.

Example: Likelihood

Likelihood

- Maximiere die Likelihood, dass die Daten vom Modell produziert wurden (mit Parameter Θ)
 - y_i : Trainingslabel für Punkt x_i
 - \hat{y}_i : Vorausgesagtes Label für Punkt x_i (mit Modell Θ)

$$\max_{\Theta} L(y, \hat{y}) = \prod_{i=0}^n L(y_i, \hat{y}_i)$$

$$-\log \prod_{i=0}^n L(y_i, \hat{y}_i) = \sum_{i=0}^n -\log L(y_i, \hat{y}_i)$$

5.9 Optimal loss function

Für kategorische Probleme (Entscheidungen zwischen 0 und 1), ist die optimale Loss Function die Cross Entropy

$$H_i = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

$$H = -\sum_{i=0}^n (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

5.10 Multi-class problem

- In Multi-Class Klassifizierung wollen wir in eine von k verschiedenen Klassen entscheiden
- Für Training und Evaluation: Wir nutzen nicht die Labels direkt, stattdessen nutzen wir One-Hot Encoded oder kategorische Vektoren

Klasse	Vektor
0	(1, 0, 0, 0)
1	(0, 1, 0, 1)
2	(0, 0, 1, 0)
3	(0, 0, 0, 1)

- Der Output soll für jede Klasse eine Wahrscheinlichkeit annähern
- Um dies zu erreichen, nutzen wir die *Softmax* Funktion als Aktivierungsfunktion für den letzten Layer

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Minimizing a function

- Wie minimieren wir eine Funktion?
- Welche Funktionen sind einfacher zu minimieren?
- Welche Eigenschaften muss eine Funktion haben, damit wir das Minimum finden können?
- Ein Weg: Einen Punkt nehmen, ableiten und determinieren auf welcher Seite die Werte hoch- oder runtergehen, für weitere Punkte der Funktion wiederholen.

5.11 Gradient descend

Wie berechnen wir den Gradienten der Loss Funktion? (Kettenregel, Ableitung)

$$y = f_0(a_1, a_2) = f_0(f_{a_1}(x_1, x_2), f_{a_2}(x_1, x_2))$$

5.12 Back propagation

Berechnung des Gradienten durch das Netzwerk. Gradient der Loss Funktion, mit Rücksicht auf den Parameter Θ .

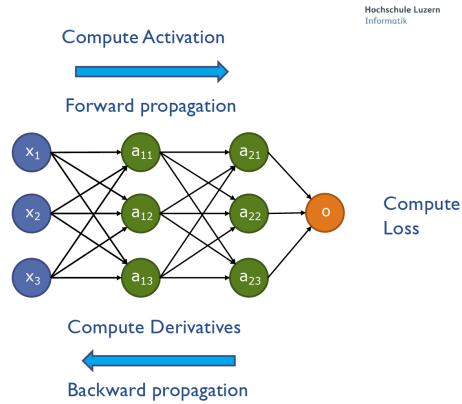


Abbildung 26: Berechnungsvorgang in einem neuronalen Netzwerk

6 Deep Neuronal Networks

7 Convolutional Neuronal Networks

8 Reinforcement Learning